Universidad Veracruzana

Research Center on Artificial Intelligence

LAAS-CNRS

Robotics, Action and Perception (RAP) group

ROS Tutorial: Robotics Operation System

Author:
Antonio Marin-Hernandez

October 31, 2014

Contents

Chapter 1

ROS Installation

1.1 Requirements

Depending on your OS, the installation of ROS require different packages and dependencies, of example:

- Linux

  ROS better works with Ubuntu 12.04 (precise).

  GCC version 4.4 or higher, CMake, Python version 2.7 (strictly).

- Mac

  Mac ports,

1.2 Installation

ROS is developed by Willow Garage Inc.^{TM} with contributions all around the world. There are more than 2,000 packages, going from diverse robotic platforms, and passing from hardware drivers (sensors and actuators) to many computer algorithms.

To begin with the installation, firs of all, setup your computer to accept software from site `packages.ros.org`. ROS Hydro ONLY supports Precise, Quantal, and Raring for Debian packages.

Ubuntu 12.04 (Precise) is preferred, so here are instructions to install on this particular distribution:

We begin adding the ROS repository by the following instruction:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise
    main" > /etc/apt/sources.list.d/ros-latest.list'
```

next, we have to set up the `keys` by:

```
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

And now, make sure your Debian package index is up-to-date:

```
$ sudo apt-get update
```

Lets proceed to install ROS, the Desktop-Full Install is recommended, it includes: ROS, rqt, rviz, robot-generic libraries, 2D/3D simulators, navigation and 2D/3D perception.

```
$ sudo apt-get install ros-hydro-desktop-full
```

Before you can use ROS, you will need to initialize `rosdep`. `rosdep` enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS.

```
$ sudo rosdep init
$ rosdep update
```

If you have already a previous version of ROS, is better to delete `rosdep` init file commonly in:

```
$ rm /etc/ros/rosdep/sources.list.d/20-default.list
```

Finally, get `rosinstall`, in ubuntu the instruction is:

```
sudo apt-get install python-rosinstall
```

1.3 Environment Setup

1.3.1 Creating a Workspace

First you need to choose a `workspace` directory, preferably an empty directory, so you can create a new one, called for example: `ros-workspace` at your home's path.

```
$ mkdir -p ~/my_workspace/src
```

from now, this workspace you have just been created, should be referred as </path/to/your/wokspace>.

```
$ cd <path/to/your/workspace>
```

This path will be the root from where `ROS` will search to find your projects.

In order to access initiate your workspace execute following lines in your terminal,

```
source <path/to/ros/install>/setup.bash
```

Now, let's create a workspace with catkin:

```
cd ~/<path/to/your/workspace>/src
catkin_init_workspace
```

Even though the workspace is empty (there are no packages in the 'src' folder, just a single CMakeLists.txt link) you can still "build" the workspace:

```
cd ~/<path/to/your/ros-workspace>
catkin_make
```

What makes directories `devel` and `build` in your </path/to/your/ros-workspace>

In order to access easily to your workspace put following lines into your `.bashrc` or `.login` file, depending on your system:

```
source <path/to/ros/install>/setup.bash
source <path/to/your/ros-workspace>/devel/setup.bash
```

To test configuration go to your home directory

```
cd
```

and type

```
roscd
```

now, you should be at

```
pwd
<path/to/your/ros-workspace>/devel
```

Chapter 2

Beginning with ROS

In this section, we will discuss about Initial concepts and tools to query system and/or communicate between modules (nodes) in ROS.

2.1 Basic Concepts in ROS

ROS is aka operating system for robots, which could be considered as a client/server system. ROS system is composed from different packages that include nodes, topics, services and parameters. Nodes, called also `rosnodes`, are the executables programs.

Topics and services are ways of communication between nodes. They depend from each node (see node definitions). Services rely on a query made by a given node or from terminal, getting a response from the node offering the service. In a different way, topics require a subscription to a node that will be broadcasting some particular info.

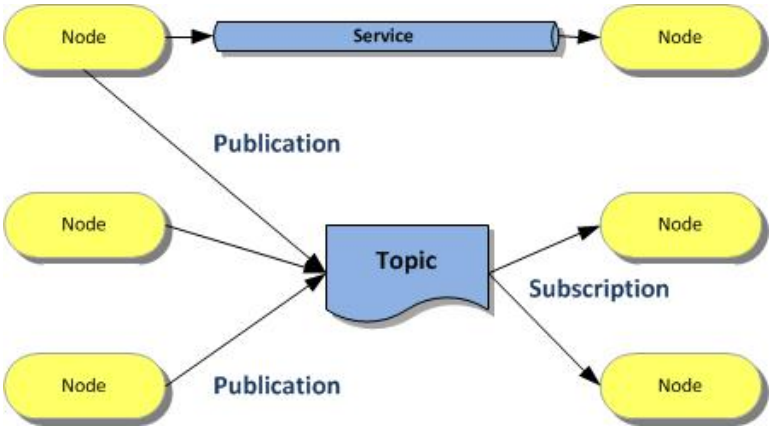

Figure 2.1: ROS concepts.

2.2 ROS Tools

ROS provides some useful tools to get information about the topics and services in run.

2.2.1 rosversion

The `rosversion` command tool, is used to recover current version in your system, by running the following command, you should get:

```
$ rosversion -d
hydro
```

unless you are using another ROS distribution (i.e. groovy or fuerte)

2.2.2 roscd

`roscd` will locate you at your ROS workspace:

```
$ roscd
$ pwd
<path/to/your/ros-workspace>/devel
```

2.2.3 rospack

You can find the path where a package is installed by:

```
$ rospack find <package>
<the/path/to/package>
```

2.2.4 rosmsg

Depending on your installed packages different types of messages can be provided. It's possible to listed by the following command:

```
$ rosmsg list
geometry_msgs/Point
geometry_msgs/Point32
geometry_msgs/PointStamped
geometry_msgs/Polygon
...
```

Following commands and tools require communication with `master` so, lets run the following command to enable master's communication:

```
$ roscore
```

Command `roscore` enables master and basic requirements and services for ROS package communication.

2.2.5 rostopic

In another terminal write the following command:

```
$ rostopic list
/rosout
/rosout_agg
```

by the moment we left this command and we will return to it later.

Chapter 3

Simulation with Gazebo

ROS includes a physics based simulation engine called Gazebo, initially developed in conjunction with the Player/Stage project.

Gazebo in ROS is considered as a node, offering different topics and services.

3.1 Requirements

To successful do this section it is required the following ROS packages:

```
gazebo_ros
turtlebot_bringup
turtlebot_description
turtlebot_gazebo
```

All these packages should be available with `apt-get` command or bias Ubuntu "Software Center".

3.2 Initializing Gazebo Simulation

To launch Gazebo with an empty world write the following command from a terminal,

```
$ roslaunch gazebo_worlds empty_world.launch
```

this will display a window with a plane.

In order to known all services offered by gazebo node we can type in another terminal window:

```
$ rosservice list gazebo
from we get the following message
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
```

```
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_gazebo_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
```

To put objects in the world there are many ways. All objects in the world are considered as robots, while some of them are not autonomous ?

Change to gazebo_worlds path by:

```
$ roscd gazebo_worlds/
```

Now, list contents in directory

```
$ ls
```

you should get something similar to:

```
CMakeLists.txt
launch
objects
test
Makefile
added_threading_stuff_test.patch
manifest.xml
scripts
worlds
Media
bin
meshes
src
```

To include an object into the scene run the following command:

```
$ rosrun gazebo spawn_model -file objects/desk1.model -gazebo -
    model desk1 -x 0
```

you should see green desk in the gazebo window.

spawn_model is a service offered by node gazebo, there are many ways to access services, in this case is by a rosrun a gazebo node with spawn_model service. spawn_model has different parameters, to known it type:

```
$ rosrun gazebo spawn_model
```

you should get something like:

```
Commands:
    -[urdf|gazebo|trimesh] - specify incoming xml is urdf or
        gazebo format
    -[file|param] [<file_name>|<param_name>] - source of the model
         xml or the trimesh file
    -model <model_name> - name of the model to be spawned.
    -reference_frame <entity_name> - optinal: name of the model/
        body where initial pose is defined.
                                    If left empty or specified as
                                         "world", gazebo world
                                        frame is used.
    -namespace <ros_namespace> - optional: all subsequent ROS
        interface plugins will be inside of this namespace.
    -unpause - optional: !!!Experimental!!! unpause physics after
        spawning model
    -wait - optional: !!!Experimental!!! wait for model to exist
    -trimesh_mass <mass in kg> - required if -trimesh is used:
        linear mass
    -trimesh_ixx <moment of inertia in kg*m^2> - required if -
        trimesh is used: moment of inertia about x-axis
    -trimesh_iyy <moment of inertia in kg*m^2> - required if -
        trimesh is used: moment of inertia about y-axis
    -trimesh_izz <moment of inertia in kg*m^2> - required if -
        trimesh is used: moment of inertia about z-axis
    -trimesh_gravity <bool> - required if -trimesh is used:
        gravity turned on for this trimesh model
    -trimesh_material <material name as a string> - required if -
        trimesh is used: E.g. Gazebo/Blue
    -trimesh_name <link name as a string> - required if -trimesh
        is used: name of the link containing the trimesh
    -x <x in meters> - optional: initial pose, use 0 if left out
    -y <y in meters> - optional: initial pose, use 0 if left out
    -z <z in meters> - optional: initial pose, use 0 if left out
    -R <roll in radians> - optional: initial pose, use 0 if left
        out
    -P <pitch in radians> - optional: initial pose, use 0 if left
        out
    -Y <yaw in radians> - optional: initial pose, use 0 if left
        out
    -J <joint_name joint_position> - optional: initialize the
        specified joint at the specified value
```

lets put another object in the world by typing :

```
$ rosrun gazebo spawn_model -file objects/000.580.67.model -gazebo
    -model cup  -z 2
```

Chapter 4

Creating Packages in ROS with catkin

4.1 Requirements

For a package to be considered a catkin package it must meet a few requirements:

The package must contain a catkin compliant `package.xml` file. That `package.xml` file provides meta information about the package. The package must contain a `CMakeLists.txt` which uses catkin. Catkin metapackages must have a boilerplate `CMakeLists.txt` file.

There can be no more than one package in each folder. This means no nested packages nor multiple packages sharing the same directory. The simplest possible package might look like this:

```
my_package/
  CMakeLists.txt
  package.xml
```

4.2 Creating a first package

To create a package you need to be at `</path/to/your/workspace>/src` directory. If it's your first package, your `src` directory should contain only a `CMakeLists.txt` file. If is not your first package it should contain a directory for each previous package. And finally if it is empty you should initialize your workspace as has been described in chapter 2.

Lets create our first package by writing the following command:

```
$ catkin_create_pkg my_first_rospkg std_msgs rospy roscpp
Created file my_first_rospkg/package.xml
Created file my_first_rospkg/CMakeLists.txt
Created folder my_first_rospkg/include/my_first_rospkg
```

```
Created folder my_first_rospkg/src
Successfully created files in </path/to/your/workspace>/src/
    my_first_rospkg. Please adjust the values in package.xml.
```

Now you have inside your `src` directory a new one called `my_first_rospkg`. And inside latter, you should have some files and directories just created.

We have just created the new package called `my_first_rospkg` which depends on : `std_msgs rospy & roscpp` .

The `catkin_create_pkg` instruction has the following format:

```
# catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

Using command `rospack` we can recover dependencies of a given package, for example:

```
$ rospack depends1 my_first_rospkg
roscpp
rospy
std_msgs
```

The previous list returned by `rospack` command, are the primarily dependencies. However as each dependencies have also their dependencies, our package has indirect dependencies. They can be recovered by:

```
$ rospack depends my_first_rospkg
catkin
console_bridge
cpp_common
rostime
roscpp_traits
roscpp_serialization
genmsg
genpy
message_runtime
gencpp
genlisp
message_generation
rosbuild
rosconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
rospack
roslib
rospy
```

We can individually recover the indirect dependencies by using same command on each of the primarily dependencies.

4.3 Compiling your Package

At this stage you can compile your new package and running it. However it doesn't contains nothing in their code, so it should do anything. To compile your package you need to be at your `</path/to/your/workspace>` and not in the `src` directory. Then you should run the command:

```
$ catkin_make
Base path: </path/to/your/workspace>
Source space: </path/to/your/workspace>/src
Build space: </path/to/your/workspace>/build
Devel space: </path/to/your/workspace>/devel
Install space: </path/to/your/workspace>/install
####
#### Running command: "cmake </path/to/your/workspace>/src -
    DCATKIN_DEVEL_PREFIX=</path/to/your/workspace>/devel -
    DCMAKE_INSTALL_PREFIX=</path/to/your/workspace>/install" in
    "</path/to/your/workspace>/build"
####
-- Using CATKIN_DEVEL_PREFIX: </path/to/your/workspace>/devel
-- Using CMAKE_PREFIX_PATH: </path/to/your/workspace>/devel;/opt/
    ros/hydro
-- This workspace overlays: </path/to/your/workspace>/devel;/opt/
    ros/hydro
-- Using PYTHON_EXECUTABLE: /usr/bin/X11/python
-- Using Debian Python package layout
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: </path/to/your/workspace>/build/
    test_results
-- Found gtest sources under '/usr/src/gtest': gtests will be
    built
-- catkin 0.5.81
-- BUILD_SHARED_LIBS is on
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 1 packages in topological order:
-- ~~  - my_first_rospkg
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'my_first_rospkg'
-- ==> add_subdirectory(my_first_rospkg)
-- Configuring done
-- Generating done
-- Build files have been written to: </path/to/your/workspace>/
    build
####
#### Running command: "make -j4 -l4" in "</path/to/your/workspace
    >/build"
####
```

In opposite way as `cmake` command works, catkin compiles everything inside the workspace a not only one package. NOTE: You should always run `catkin_make` command from your root workspace, i.e. <path/to/your/ros-workspace>.

4.4 Customizing your Package

Lets begin `gazebo` simulator with the turtlebot robot by tapping on your `teminal`:

```
$ roslaunch turtlebot_gazebo turtlebot_empty_world.launch
```

Turtlebot uses the following topic `/mobile_base/commands/velocity` to get speed commands. This topic is of the type `geometry_msgs/Twist`, as can be see using the `rostopic info` command. In order to communicate with our package with the robot, it's necessary to create a code able to write on this topic.

Lets add this code to our package. Open your favorite code editor, e.g. `emacs` or `gedit`, and create a file called `my_first_code.cpp` in the `src` directory of `my_first_rospkg` and then copy following lines inside it:

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_fisrt_node");
  ros::NodeHandle n;
  ros::Publisher speed_pub = n.advertise<geometry_msgs::Twist>("/
      mobile_base/commands/velocity", 1000);
  ros::Rate loop_rate(10);

  int count = 0;

  while (ros::ok()) {
    geometry_msgs::Twist speedMsg;
    speedMsg.linear.x=0.0;
    speedMsg.linear.y=0.0;
    speedMsg.linear.z=0.0;
    speedMsg.angular.x=0.0;
    speedMsg.angular.y=0.0;
    speedMsg.angular.z=0.5;

    speed_pub.publish(speedMsg);

    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

Now, lets modify `CMakeLists.txt` file on the `home` of your package, by uncomment and modifying or writing the following lines:

```
## Declare a cpp executable
add_executable(my_first_rospkg_node src/my_first_code.cpp)
```

and

```
## Specify libraries to link a library or executable target
    against
 target_link_libraries(my_first_rospkg_node
   ${catkin_LIBRARIES}
 )
```

Now lets compile the node by running `catkin_make` command on the top of your ros workspace. At this stage you are able to make move your simulated robot on gazebo by running the command:

```
$ rosrun my_first_package my_first_rospkg_node
```

4.5 Reading and Writing Topics in your code

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"

ros::Publisher speed_pub;
geometry_msgs::Twist speedMsg;

void poseCallback(const nav_msgs::Odometry::ConstPtr& odomMsg)
{

  ROS_INFO("Turtlebot -> Reading Odometry Message %f,%f\n",
   odomMsg->pose.pose.position.x,
   odomMsg->pose.pose.position.y);

    speedMsg.linear.x=0.5;
    speedMsg.linear.y=0.0;
    speedMsg.linear.z=0.0;
    speedMsg.angular.x=0.0;
    speedMsg.angular.y=0.0;
    speedMsg.angular.z=0.0;

    if ((fabs(odomMsg->pose.pose.position.x)>2.00) ||
  (fabs(odomMsg->pose.pose.position.y)>2.00)){
      speedMsg.linear.x=-0.5;
    }
     speed_pub.publish(speedMsg);
}



int main(int argc, char **argv)
{
  ros::init(argc, argv, "publisher");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("/odom", 1000,poseCallback);
```

```
  speed_pub = n.advertise<geometry_msgs::Twist>("/mobile_base/
      commands/velocity", 1000);

  printf("P3DX Reader initialized\n");
  ros::spin();
  return 0;
}
```

4.6 Exercises

4.6.1 Circles

Make your simulated robot to move describing a circle of radius $r = 1.0m$

4.6.2 Square

Following a square of $1m$ by side

4.6.3 Spiral

Chapter 5

Creating your Own Robot Model

5.1 URDF models

5.2 Robot State Publisher

5.3 Joint States

Chapter 6

References